DEPARTAMENTO DE COMPUTACION

Facultad de Ciencias Exactas y Naturales - UBA

# Primer Trabajo Práctico

14 de Abril de 2010 — Algoritmos y Estructuras de Datos III


| Integrante | LU | Correo electrónico |
|---|---|---|
| Bianchi, Mariano | 92/08 | bianchi-mariano@hotmail.com |
| Brusco, Pablo | 527/08 | pablo.brusco@gmail.com |
| Di Pietro, Carlos Augusto Lyon | 126/08 | cdipietro@dc.uba.ar |





**Facultad de Ciencias Exactas y Naturales**

Universidad de Buenos Aires

Ciudad Universitaria - (Pabellón I/Planta Baja)

Intendente Güiraldes 2160 - C1428EGA

Ciudad Autónoma de Buenos Aires - Rep. Argentina

Tel/Fax: (54 11) 4576-3359

http://www.fcen.uba.ar

# Índice

# 1. Ejercicio 1

## 1.1. Introducción

El primer problema del presente trabajo consistió en la implementación de un algoritmo capaz de dar solución a la ecuación

$$b^n \ mod \ (n) \tag{1}$$

haciendo uso de alguna de las técnicas algorítmicas aprendidas hasta el momento en la materia. Asímismo, la consigna dictaba que la complejidad final del algoritmo debería ser menor a $O(n)$.

En pos de cumplimentar lo pedido se decidió usar la técnica de *Dividir & Conquistar*[1] para desarrollar el algoritmo. Esta técnica se caracteriza principalmente en dividir la instancia de un problema en instancias más pequeñas, atacar cada una de ellas por separado y resolverlas, para finalmente juntar sus resultados y así producir el resultado final.

## 1.2. Explicación

La primera solución que se piensa casi de manera intuitiva para el problema es la de mutliplicar $n$ veces el número $b$ y luego hallar el resto de dividir ese resultado por $n$.

$$\underbrace{b.b.b \ldots b.b.b}_{n \ veces} \ mod \ (n) = b^n \ mod \ (n) \tag{2}$$

Sin embargo, salta a la vista que la complejidad ese algoritmo es $O(n)$, ya que se realizan $n$ multipicaciones y 1 división, razón por lo cual no cumple con lo pedido en la consigna. Además, puesto que ese algoritmo realizar sucesivas multipliciones de $b$, cabe la posibilidad de que el valor que se va acumulando sobrepase el máximo número entero representable en la computadora produciedo así un overflow y obteniéndose en consecuencia un resultado incorrecto.

Analizando la falla del algoritmo anterior, se observa que lo que se debe evitar es calcular directamente el resultado de $b^n$ ya que ese número podría ser excesivamente grande.
Con esto en mente veamos una manera más conveniente de abordar el problema. Es claro, por definición de congruencia, que resover la ecuación en (1) equivale a hallar el $x$ tal que:

$$b^n \ \equiv \ x \ (n) \ , \quad con \ 0 \ \leq \ x \ < \ n \tag{3}$$

Luego, asumamos por un momento que podemos tener gratis un k tal que:

$$b^{n/2} \ \equiv \ k \ (n) \ , \quad con \ 0 \ \leq \ k \ < \ n \tag{4}$$

Entonces, aplicando el resultado de (4) en (3) obtenemos que:

- Si $n$ es par:

$$\begin{aligned} b^n \ \equiv \ x \ (n) \ , \quad con \ 0 \ \leq \ x \ < \ n \\ b^{n/2}.b^{n/2} \ \equiv \ x \ (n) \\ k.k \ \equiv \ x \ (n) \\ k^2 \ \equiv \ x \ (n) \end{aligned} \tag{5}$$

[1] Poner alguna referencia en donde se explique esta técnica

- Si $n$ es impar par:

$$\begin{aligned} b^n \equiv x\ (n)\ , \quad con\ 0 \leq x < n \\ b.b^{n/2}.b^{n/2} &\equiv x\ (n) \\ b.k.k &\equiv x\ (n) \\ b.k^2 &\equiv x\ (n) \end{aligned} \tag{6}$$

Luego, las expresiones resultantes en ambos casos pueden resolver de manera más simple y rápida ya que al estar acotado $k$ por $n$ el valor de $b.k^2$ es mucho menor que el de $b^n$ y en consecuencia la operación de calcular el resto módulo $n$ se realiza en menor tiempo.

Volviendo un poco sobre nuestro pasos, se dijo anteriormente en (4) que podíamos asumir que obtener $k$ no tenía un costo computacional significativo. No obstante, esto no siempre es cierto puesto que para obtener $k$ es necesario calcular previamente $b^{n/2}$ el cual puede ser un número nada despreciable en lo que respecta a tamaño en memoria.
Si ese fuera el caso, entonces para calcular $b^{n/2}$ podemos aplicar nuevamente el procedimiento descripto con anterioridad calculando $b^{n/4}$ y usando una expresion análoga a (5) o a (6) dependiendo de si $n/2$ es par o impar respectivamente. Del mismo modo se puede repetir el accionar con $b^{n/4}$ y así sucesivamente hasta reducir el exponente a 1, donde $b^1$ *mod* $n$ puede calcularse en O(1) ya que $b$ es acotado. A partir de allí se realiza el camino inverso calculando cada uno de los restos módulo $n$ de las potencias de $b$ a partir del resto de dividir por $n$ a la potencia de $b$ calculada en el paso inmediatamente anterior, hasta dar finalmente con la solución del problema.

## 1.3. Análisis de la complejidad del algoritmo

A continuación se calculará la complejidad del algortimo implementado en la función *potenciacion*, que es el que resuelve el problema que se planteo en (1). Dicho problema tiene por variables a $b \in \mathbb{N}$ y $n \in \mathbb{N}$, lo cual en términos matemáticos representa que $b$ y $n$ pueden tomar valores tan grandes como se desee puesto que $\mathbb{N}$ es un conjunto infinito. Esto se ve traducido en que los parámetros del algoritmo *potenciacion* sean tan grandes como uno desee.
Ahora bien, la consigna asegura que $b$ puede considerarse como un número acotado; pero aún así, $n$ puede seguir siendo tan grande como se quiera. Por este motivo, resulta lógico pensar que la suposición de que $n$ cabe perfectamente en una posicion fija de memoria no es para nada correcta. Por lo tanto, para que el análisis de complejidad del algoritmo sea lo más correcto posible se debe evaluar la complejidad de las operaciónes que realiza el algoritmo en función de la cantidad de símbolos binarios necesarios para representar el valor de cada parámetro dentro del ordenador; en particular en función del tamaño de n, ya que es el único parámetro no acotado.
Lo expresado en el análisis anterior conlleva a que la medición de complejidad del algortimo se efectúe mediante el *modelo logarítmico* el cual precisamente mide la performance de un algoritmo a través de la suma de los costos de todas las operaciones en función del tamaño de las variables involucradas.

Los costos de las operaciones elementales en el modelo logarítmico son representadas en la siguiente tabla:

| operaciones | costo en m. logarítmico |
|---|---|
| m+n | $log_2$ (m) + $log_2$ (n) |
| m-n | $log_2$ (m) + $log_2$ (n) |
| m*n | $log_2$ (m) + $log_2$ (n) |
| m/n | $log_2$ (m) + $log_2$ (n) |
| [ i ] | $log_2$ (i) |
| a=n | $log_2$ (n) |

Sigue a continuación un pseudocódigo del algoritmo en cuestión en el que se detallan, para cada instrucción, su costo en el modelo logarítmico:

*potenciacion(base :* $\mathbb{N}$*, exp:* $\mathbb{N}$*, m:* $\mathbb{N}$*)*

```
1  Complejidad: T(n)
2  var res : ℤ
3  base ← base mod m ;                                          // O(log2(base))
4  if (base == 0);                                              // O(log2(base))
5  then
6  |  return base
7  else
8  |  if (exp == 1);                                            // O(log2(exp))
9  |  then
10 |  |  return base
11 |  else
12 |  |  if (exp mod 2 == 0) ;                                  // O(log2(exp))
13 |  |  then
14 |  |  |  var temp : ℤ
15 |  |  |  temp ← potenciacion(b, exp/2, m) ;                  // T(exp/2)
16 |  |  |  temp ← (temp*temp) ;                               // O(log2(temp)^2)
17 |  |  |  temp ← temp mod m ;                                // O(log2(temp) * log2(m))
18 |  |  |  res ← temp ;                                       // O(log2(temp))
19 |  |  else
20 |  |  |  var temp : ℤ
21 |  |  |  temp ← potenciacion(b, (exp-1)/2, m) ;             // T((exp-1)/2)
22 |  |  |  temp ← (temp*temp) ;                               // O(log2(temp)^2)
23 |  |  |  temp ← temp mod m ;                                // O(log2(temp) * log2(m))
24 |  |  |  temp ← temp * base ;                               // O(log2(temp) * log2(b))
25 |  |  |  temp ← temp mod m ;                                // O(log2(temp) * log2(m))
26 |  |  |  res ← temp ;                                       // O(log2(temp))
27 |  |  end
28 |  end
29 end
30 return res
```

**Algorithm 1**: Pseudocódigo de la función *potenciación* con el costo de cada instrucción en el modelo logarítmico

Como la función es recursiva, sería correcto analizar su complejidad remitiéndonos a las complejidades que se desprenden del teorema maestro. Sabemos que la forma del caso recursivo es:

$$T(n) = a * T(n/b) + f(n) \tag{7}$$

donde a es la cantidad de llamados recursivos, b es la cantidad de subproblemas y f(n) es el costo de cada paso sin considerar las llamadas recursivas.
En el caso de nuestro algoritmo, a = 1 y b = 2. Si observamos detalladamente las complejidades marcadas en el pseudocódigo, podemos ver que la complejidad mayor se da cuando se vuelve

del llamado recursivo, en el momento que se eleva al cuadrado la variable *temp*. Esta variable vale como máximo m-1. Por lo tanto, $f(n) = log^2(n)$. Además, f(n) vale lo mismo sin importar la entrada, por lo que $f(n) \in \Theta(log^2(n))$

De esto se desprende entonces que para el caso de nuestro algoritmo, el modelo de complejidad recursiva esta dado por:

$$T(m) = T(m/2) + f(log^2(n)) \tag{8}$$

Por otra parte, sabemos por el teorema maestro que:

$$Si\ f(n) \in \Theta(n^{log_b(a)} log^k(n)) \Rightarrow T(n) \in \Theta(n^{log_b(a)} log^{k+1}(n)) \tag{9}$$

Si en (11) aplicamos los valores de a y b antes mencionados, vemos que:

$$f(n) \in \Theta(n^{log_2(1)} log^2(n)) = \Theta(log^2(n)) \tag{10}$$

Como $f(n) = \Theta(log^2(n))$, por (11 puedo decir finalmente que la cota teórica del algoritmo implementado es:

$$T(n) \in \Theta(log^3(n)) \tag{11}$$

Sólo resta hacer el análisis de la complejidad de acuerdo al tamaño de la entrada. El tamaño de la entrada es $E(I) = 2 * log(m) > log(m)$ mientras que la complejidad de una instancia es $C(I) = log^3(m)$.
Ahora, con estos valores podemos ver que:

$$C(I) \leq c' * log^3(m) \leq c' E(I)^3 \tag{12}$$

Entonces,

$$C(I) \leq c' E(I)^3 \Leftrightarrow C(I) \in O(E(I)^3) \tag{13}$$

## 1.4. Detalles de Implementación

Dentro de la carpeta *./ej1/*, se puede encontrar el archivo ejecutable *ejercicio_1* compilado para GNU-Linux, el cual resuelve el problema anteriormente descripto. Este programa se ejecuta por consola mediante el comando `./ejercicio_1`, y recibe como parámetro los archivos de entrada *".in"* a procesar. Puede recibir tantos nombres de archivo como se desee, pero en caso de no recibir ninguno, el programa procesará el archivo *Tp1Ej1.in* que se encuentra incluído dentro de la misma carpeta.
Una vez ejecutado, el algoritmo procesa la cola de archivos que recibió como parámertos de a uno por vez generando para cada uno de ellos dos archivos:

- Un archivo *".out"* omónimo con la solucion a la ecuación (1) para cada par de naturales (b, n) contenidos en el archivo de entrada. Este archivo se guarda en la misma carpeta en la que se encuentra el ejecutable *ejercicio_1*.
- Un archivo omónimo con el sufijo *"_grafico.out"*, en el cual registra para cada n el número de operaciones realizadas por el algoritmo. Este archivo se guarda en la carpeta *./ej1/info graficos*, y tiene por objetivo facilitar la tarea de cargar los datos en el programa de análisis gráfico *QtiPlot*.

Por otra parte, en la misma carpeta, se puede hallar el archivo ejecutable *input_gen1* también compilado para GNU-Linux. Al correr este programa desde la consola mediante el comando `./input_gen1` se despliega un menú de opciones para generar distintos tipos de archivos *".in"* para ser resueltos por el ejecutable *ejercicio_1*. Una vez elegido el tipo de entrada a crear, el programa solicita que se ingrese un nombre de archivo y la cantidad de casos a generar. Acto seguido guarda el archivo generado en la carpeta *./ej1/*.

Asímismo, en la carpeta *./ej1/* hay un Makefile el cual permite recompilar los archivos ejecutables con tan solo ejecutar el comando `make` en la consola. Además, ejecutando el comando `make clean` se pueden eliminar los archivos ejecutables y todos los archivos de extensión *".out"*.

Luego, en la carpeta *./ej1/sources* se encuentran los codigos fuentes de los ejecutables antes descriptos. Los mismo están escritos en lenguaje C++ y tienen comentadas las partes relevantes para simplificar la comprensión.

Finalmente, en *./ej1/* se hayan los archivos *.Tp1Ej1.in* y *Tp1Ej1.out* que vienen junto con en el enunciado de este Trabajo Práctico, y se haya también la carpeta *./ej1/test* la cual contiene los archivos *".in"* generados para probar el algortimo junto con sus correspondientes gráficos de $n$ vs. *cantidad de operaciones*.

### 1.5. Debate

Como primera observación de los gráficos expuestos en la sección anterior, cabe decir que la complejidad real se ajustó a la complejidad teórica, ya que en todos los casos se pudo hallar una constante $c \in \mathbb{R}$ tal que a partir de algún $n_0$ el gráfico de $c.log_2(\mathrm{n})$ pasa a acotar superiormente a la cantidad de operaciones realizadas por el algoritmo.

En lo que respecta a la las hipótesis postuladas en la sección 4.1, analicemos una a una su veracidad en función de los gráficos obtenidos.
Se puede advertir mediante la observación del 3 que se cumple la hipótesis 1, ya que para todo n la cantidad de operaciones es constante. Del mismo módo, cotejando 1 y 2 se corrobora la hipótesis 2 ya que para los *n's* del caso `a` la cantidad de operaciones asciende hasta 320 aproximadamente, mientras que en el caso `b` no traspasan la barrera de las 250 operaciones.
En cuanto a la hipótesis 3, la comparación entre 1 y **??** no pareciera arrojar ningún dato que pudiera confirmar la suposición. Más aún, pareciera refutarla, ya que para el grueso de los *n's* medidos en cada caso, el algoritmo encuentra la solución al problema en 300 operaciones aproximadamente.
Por último, en lo que concierne a la hipótesis 4, viendo el 3 se ve que para *n's* en el orden de $10^9$, el algoritmo llega a realizar alrrededor de 340 operaciones; valor que sin duda supera a todas las cantidades de operaciones de los demás casos. Puede verse también que la distribución de las mediciones es la que mejor se ajusta a la curva logarítmica de entre las restantes mediciones.

## 1.6. Conclusiones

En base analizado en la sección 1.5, es digno de remarcar el como conclusión hecho de que la complejidad medida en la realidad se asemeja bastante a las complejidades teóricas estimadas. Asímismo, el hecho de que el número de operaciones necesarias para resolver el algoritmo crece de manera logarítmica conforme el tamaño de la entrada aumente. No obstante, en el caso en que $b$ sea un múltiplo de $n$, el algoritmo será capaz de resolver el problema en un número constante de operaciones, aunque no así de tiempo, ya que el tiempo que demore en realizar las operaciones dependerá del tamaño de $n$. Por último, se puede concluir que el hecho de que $n$ sea un número primo no conlleva a un aumento significativo en la cantidad de operaciones, por lo que la complejidad real en esos casos se mantiene consistente a la complejidad teórica.

# 2. Ejercicio 2

## 2.1. Introducción

El segundo problema consistió en la implementación de un algoritmo capaz de dar solución al problema que se plantea a continuación:

Decidir si un grupo de n personas pueden formar o no una ronda que cumpla con las siguientes restricciones:

- La ronda debe contener a todas las personas.
- Algunas personas son amigas y otras no.
- Cada alumna debe tomar de la mano a dos de sus amigas.

## 2.2. Explicación

Dado que para este problema no se conocen algoritmos buenos[2], se pensó en utilizar la solución por fuerza bruta, es decir, intentar todas las combinaciones hasta lograr determinar si hay o no solución, pero, agregando algunas mejoras.

Mejoras como evaluar, en el momento de cargar la ronda, si alguna persona no tiene las suficientes amigos, es decir 2, o si todos son amigos de todos.

Luego, se pensó en la estrategia de "Vuelta atrás", (Backtracking).

***Backtracking:***
*En su forma básica, la idea de backtracking se asemeja a un recorrido en profundidad dentro de un grafo dirigido. El grafo en cuestión suele ser un árbol, o por lo menos no contiene ciclos. Sea cual sea su estructura, existe sólo implícitamente. El objetivo del recorrido es encontrar soluciones para algún problema. Esto se consigue construyendo soluciones parciales a medida que progresa el recorrido; estas soluciones parciales limitan las regiones en las que se puede encontrar una solución completa. El recorrido tiene éxito si, procediendo de esta forma, se puede definir por completo una solución. En este caso el algoritmo puede bien detenerse (si lo único que se necesita es una solución del problema) o bien seguir buscando soluciones alternativas (si deseamos examinarlas todas). Por otra parte, el recorrido no tiene éxito si en alguna etapa la solución parcial construida hasta el momento no se puede completar. En tal caso, el recorrido vuelve atrás exactamente igual que en un recorrido en profundidad, eliminando sobre la marcha los elementos que se hubieran añadido en cada fase. Cuando vuelve a un nodo que tiene uno o más vecinos sin explorar, prosigue el recorrido de una solución.* [3]

En este caso particular, la idea es, seleccionar una persona (P), ingresarla en la ronda, e ir armando un arbol implicito de posibilidades en donde el nodo padre es P y cada rama se forma con las distintas selecciones de amigas de P y sus sub-arboles, con las combinaciones posibles de amigas del nodo padre en el que esté. Para ello se utiliza la recursión, en donde se

[2] Un algoritmo se considera bueno si puede ser resuelto en tiempo polinomial.
[3] http:es.wikipedia.org

intenta bajar lo más posible en el arbol hasta llegar a formar un camino desde el primer nodo hasta alguna hoja en donde se encuentren todas las chicas con nodos padre e hijo amigas.

También, se implementaron otras mejoras que intentan podar el arbol dadas ciertas condiciones que permitan acortar las posibilidades:

- Las amigas de un nodo se pueden procesar en cualquier orden, pero decidimos intentar ser lo más restrictivo posible con las primeras (las que tienen menos amigas primero). Este proceso poda el árbol de búsqueda antes de que se tome la decisión y se llame a la subrutina recursiva.
- Cuando se elige qué valor se va a asignar, se hace un examen comprobando que aún queden amigas de la que inició la ronda disponibles. En caso que no, se pueden evitar muchas llamadas sin sentido.

A continuación se ve un pseudocodigo que muestra el comportamiento de la estrategia y las mejoras elegidas.

Algunos detalles:

- **gente** es una lista con elementos de tipo chica, ordenadas de acuerdo a la cantidad de amigas que tiene cada elemento.
- el tipo **chica** consta de un nombre (int) y sus amigas (conjunto de nombres).
- **enRonda** es un conjunto de nombres donde se va almacenando quienes pertenecen a la ronda.
- **amigasPrimera** es un conjunto de nombres donde se almacenan los nombres de las amigas de la chica que comienza la ronda.
- **tam(c)** devuelve el tamaño del conjunto c.
- **insertar(c,e)** inserta a e en el conjunto c.
- **borrar(c,e)** borra a e del conjunto c.
- **cuenta(c,e)** cuenta la cantidad de apariciones del elemento e en c.
- **primerElemento(l)** devuelve el primer elemento de una lista.

**resolver()**

```
1  Complejidad: T(n)
2  var n : int ← tam(gente)
3  var sonPocas : bool ← false
4  var sonSuficientes : bool ← false
5  foreach k en gente do
6  |  sonPocas ← sonPocas or tam(amigas(k)) < 2;
7  |  sonSuficientes = sonSuficientes && tam(amigas(k)) == n-1);
8  end
9  if (sonSuficientes) then
10 |  retornar true
11 end
12 if (sonPocas) then
13 |  retornar false
14 end
15 var solitaria : chica ← primerElemento(gente);
16 insertar(nombre(solitaria),enRonda);
17 retornar probarDistintasRondas(solitaria,solitaria);
```

**probarDistintasRondas(prim : chica, ult : chica)**

```
1  Complejidad: T(n)
2  if (cuenta(amigas(ult),nombre(prim)) ≠ 0 && tam(gente) = tam(enRonda)
   ) then
3  |  retornar true;
4  else
5  |  var res : bool
6  |  var eraAmiga : bool ← false
7  |  foreach chi en gente do
8  |  |  var estaEnRonda : int ← cuenta(enRonda,nombre(chi))
9  |  |  var esAmigaDeLaUltima : int ← cuenta(amigas(ult),nombre(chi))
10 |  |  var restantesAmigasPrimera : int ← tam(amigasPrimera)
11 |  |  var esAmigaDeLaPrimera : int ← cuenta(amigas(prim),nombre(chi))
12 |  |  if ( (estaEnRonda = 0) && (esAmigaDeLaUltima ≠ 0) &&
   |  |  (restantesAmigasPrimera ≠ 0) ) then
13 |  |  |  if (esAmigaDeLaPrimera ≠ 0) then
14 |  |  |  |  eraAmiga ← true;
15 |  |  |  |  eliminar(amigasPrimera,nombre(chi));
16 |  |  |  else
17 |  |  |  |  eraAmiga ← false;
18 |  |  |  end
19 |  |  |  insertar(enRonda,nombre(chi));
20 |  |  |  res ← probarDistintasRondas(prim,chi);
21 |  |  |  if (res) then
22 |  |  |  |  retornar true;
23 |  |  |  else
24 |  |  |  |  borrar(enRonda,nombre(chi));
25 |  |  |  |  if (eraAmiga) then
26 |  |  |  |  |  insertar(amigasPrimera,nombre(chi))
27 |  |  |  |  end
28 |  |  |  end
29 |  |  end
30 |  end
31 |  retornar false;
32 end
```

## 2.3. Análisis de la complejidad del algoritmo

La complejidad en el peor caso de este algoritmo surge inmediatamente al hacer pruebas. Dependiendo de las entradas elegidas, el programa podría tener que recorrer casi todas las ramas del árbol hasta poder decir que no hay solución. En este caso, podríamos entonces solo acotar el peor caso por la función $f(n) = c.(n-1)! + b$ (donde b y c son constantes) ya que

la cantidad de formas de armar una ronda con n personas es (n!/n) = (n-1)![4]

Por lo tanto, el algoritmo implementado NO es bueno. Sin embargo, cuando para un conjunto de datos de entrada se puede formar una ronda, el algoritmo no recorre todo el árbol en busca de la solución, sino que al encontrar una termina. Esta propiedad del backtracking y el resto de las mejoras, hace que para ciertos datos de entradas en donde se pueden formar rondas, el algoritmo se comporte como lo haría una solución polinomial.

En lo que respecta a la complejidad del mejor caso, se puede ver de forma simple que el mismo se da si en el primer intento de formar una ronda se lograra llegar a una respuesta válida. Esto significa que el algoritmo demoró $n$ pasos para unir a las amigas y uno más para verificar que era una ronda válida. Por lo tanto, la complejidad en el mejor caso es $\Omega(n)$ [5]

## 2.4. Detalles de Implementación

Para compilar este programa, sólo hace falta ejecutar el comando make. El modo de uso del mismo es simple: En la carpeta que contiene al ejecutable debe existir un archivo con los datos de entrada llamado "entrada". El programa pasará a ejecutarse al escribir .main en la consola y una vez que termine, devolverá en un archivo llamado "salida" el resultado buscado.

Si se desean generar casos de pruebas, se puede utilizar el generador que se encuentra dentro de la carpeta del ejercicio 2, en la subcarpeta otros.

Este generador se puede compilar utilizando el comando "g++ main.cpp" en la consola de linux, y luego ejecutandolo con el comando .a.out

Una vez abierto, el programa solicita un porcentaje ($0 \leq porc \leq 100$) y un numero maximo de chicas (max), el funcionamiento es el siguiente:

Se generan rondas de tamaño 2, 3, 4, ..., max inclusive, en donde en cada una, el programa intenta formar aproximadamente $(porc/100) * numeroTotalDeConecciones$ de amistades. Es decir, para $n$ chicas, hay $n.(n-1)/2$ rondas amistades posibles, por lo tanto, el programa intentara formar $(porc/100) * (n * (n-1)/2)$ amistades en la ronda.

Este programa genera un archivo con el nombre "entradap", que puede ser utilizado como entrada para el programa principal (cambiandole el nombre por "entrada").

Si se desea medir el tiempo que demora el algoritmo para una entrada, se puede utilizar el programa almacenado en la carpeta Ej2otrosTiempos.

Los pasos a seguir son equivalentes a los del programa original, necesita una entrada "entrada" y ser compilado con la instruccion make. luego con .main

## 2.5. Debate

De los resultados pertenecientes a la sección [2.3] podemos observar cómo se comporta la implementación frente a algunas entradas particulares. Las mismas fueron creadas al azar

[4] Se lo puede imaginar como una fila de n personas. Para la primer posición hay n personas disponibles, para la segunda n-1, y así sucesivamente. Por lo tanto hay n*(n-1)*...*1 = n! formas de acomodar las personas en una fila. Ahora, si se unen las puntas, podría rotar la fila n veces y seguiría siendo la misma ronda, por lo tanto dividiendo por n obtengo la respuesta: **(n-1)!**.

[5] esta cota menor se podría dar optimizando el algoritmo para que se comporte de la manera descripta pero sin algunas mejoras como comprobar para adelante en cada paso por ejemplo.

pero de forma tal que se cumplan los invariantes citados en [2.1] y con el agregado de fijar la cantidad de enlaces (amistades) en cada ronda (esta función del generador de casos de prueba es explicado en la sección **detalles de implementación**.

En un primer análisis, podemos ver que en general para todos los casos analizados la función proveniente de graficar cantidad de chicas contra tiempo demorado en ejecutar el algoritmo se asemeja bastante a una función polinomial cuando la cantidad de relaciones es alta. Si bien sabemos que esto no refleja la verdadera complejidad del algoritmo, en primer instancia podría significar un buen funcionamiento de las mejoras realizadas sobre el algoritmo.

La función antes mencionada, pareciera ser bastante monótona. Igualmente, se observa con claridad la aparición de casos en los cuales se demoró mucho más tiempo que el tiempo promedio. Esto podría dejar entrever que para ciertos valores de la entrada o no se encontró resultado alguno, por lo cuál se tuvieron que analizar los (n-1)! casos posibles, o bien se encontró una solución al problema, pero esta se demoró en ser encontrada ya que se trataba de un ejemplo en el cual las mejoras realizadas no surtieron efecto o, peor aún, tuvieron un efecto contrario al deseado.

## 2.6. Conclusiones

Como resultado del análisis de los datos arrojados en [4.2] y de las hipótesis realizadas anteriormente, podemos concluir:.

Primero, para muchos de los casos analizados durante el transcurso de este trabajo práctico, el algoritmo llegó a una respuesta en un tiempo promedio mucho menor al que en teoría debería demorar. Por lo tanto, podemos intuir que las mejoras realizadas dentro del algoritmo fueron, en general, muy productivas. Sin embargo, no nos aseguran que el algoritmo mejore su complejidad asintóticamente.

En segundo lugar, y continuando lo antes dicho, se observaron casos en los cuales el algoritmo demoró más que el tiempo promedio. Esto puede deberse a que, para ciertos casos de entrada, las mejoras del algoritmo no inciden en la resolución del problema, haciendo que se demore demasiado en conseguir una solución válida. Si se suma el hecho de analizar dichos casos para entradas de gran tamaño, podemos llegar a estar ante un problema, ya que el tiempo de resolución del problema para nuestro algoritmo puede ser, en términos humanos, eterno.

Finalmente, podemos concluir entonces que el algoritmo, más allá de las mejoras implementadas sobre el mismo, no alcanza a obtener una complejidad de peor caso menor que $O(n!)$, pero las mismas sirven para que el algoritmo encuentre más fácilmente el camino hacia una buena solución[6].

[6]Siempre y cuando exista al menos una solución al problema y además, esta solución se condiga con las mejoras elegidas en la implementación.

# 3. Ejercicio 3

## 3.1. Introducción

El tercer y último problema de este trabajo práctico consiste en, dadas dos listas con horarios de entrada y salida de cierto grupo de trabajadores a una empresa, se quiere saber cuál es la máxima cantidad de empleados que se encuentran en simultáneo dentro de dicha empresa. Estas listas cumplen con la particularidad de que:

- Están ordenadas ascendentemente por horario.
- Las horas van desde 00:00:00 hasta 23:59:59
- Para todos los trabajadores, es cierto que su horario de ingreso es estrictamente menor que su horario de egreso.

Además, como parte de la consigna se solicitaba que el algoritmo produjese un resultado correcto en tiempo estrictamente menor a $O(n^2)$, donde n es la cantidad de trabajadores pertenecientes a la empresa.

La solución empleada no requirió de una técnica algorítmica compleja, sino que aprovechó las características de los datos de entrada para resolver el problema de una forma eficaz. Conociendo que existen dos listas (una de horarios de entrada y otra de horarios de salidas), y que ambas listas estaán ordenadas por horarios, bastó simplemente con un recorrido lineal sobre ellas para poder llevar un registro del número de empleados dentro de la empresa y así averiguar con certeza la solución al problema.

## 3.2. Explicación

### 3.2.1. Primer aproximación

En una primer mirada, se pensó que la mejor forma de resolver el ejercicio era aplicando un algoritmo similar al quicksort[7], sólo que el mismo se realizaba sobre conjuntos y no sobre arreglos. En resumidas cuentas, la idea consistía en tomar un trabajador cualquiera del grupo para actuar de pivot, y luego mediante la comparación con el pivot, separar a los restantes en tres conjuntos distintos: los que salían antes que el pivote entrara a la empresa, los que entraban luego que el pivote saliera de la empresa y los que se cruzaban con el trabajador pivote (tanto cuando el pivote entraba y el otro salá como en el caso inverso). De esta manera, si se repetía el proceso varias veces, se obtenía varios conjuntos en los cuales aparecían únicamente los trabajadores que se cruzaban en sus horarios dentro de la empresa. Finalmente, sólo restaba devolver el mayor cardinal de dichos conjuntos.
Al igual que el quicksort, si el pivote era elegido al azar, el algoritmo anterior contaba con una complejidad promedio de $O(n * log(n))$. De cualquier modo, el peor caso seguía siendo $O(n^2)$, por lo que no se cumplía con la cota máxima pedida para este trabajo.

[7] Alguna referencia que explique el algoritmo

Como corolario de un análisis más exhaustivo del problema, se cayó en la cuenta que los datos de entrada contaban con la característica de estar ordenados por horario, tanto en la lista de entradas como en la de salidas. Dicha observación posibilitó el desarrollo de una idea capaz de solucionar el problema en tiempo $O(n)$, la cual se expone en la sección a continuacíon.

### 3.2.2. Segunda aproximación

Para encontrar solución al problema dado, lo primero que se realizó fue pensar qué cosas se necesitaban para representarlo. Para ello se creó una clase "Empresa", sobre la cual se cargarían los datos de entrada. La clase en cuestión consta de dos arreglos de tuplas $\langle$ horario, trabajador[8]$\rangle$. Dichos arreglos se hallan ordenados ascendentemente por horario; uno de los ellos contiene únicamente los horarios de entrada, mientras que el otro contiene únicamente los de salida.

Una vez resuelto el problema de cómo albergar los datos de manera de poder acceder a ellos de forma eficiente, se prosiguió con el desarrollo del algoritmo que resolviera el problema planteado en la intrducción.
La idea del mismo consiste en ir indexándo ambos arreglos desde la posición 0 hasta la n-1 ésima. De esta forma, se recorre el arreglo que contiene los horarios de entrada, hasta que se encuentra posición en la cual el horario de entrada sea mayor al horario de salida que se está indexando en ese momento (por ejemplo, en el caso de la primera iteración, el horario ubicado en la posición 0 del arreglo de horarios de salida). Mientras esto se realiza, un contador con valor inicial 0 se va incrementando con cada iteración, simulando así la cantidad de trabajadores dentro del establecimiento.
De momento que se halla un horario de entrada mayor al de salida esto significa que, antes que ingrese el trabajador correspondiente a ese horario de entrada, hubo al menos uno que se retiró. Esto lleva a dejar de iterar el arreglo de entradas y pasar a iterar el de salidas, al tiempo que se guarda en la variable a devolver el máximo entre el contador de trabajadores dentro de la empresa y el mayor número de empleados que hubo dentro de la empresa registrado hasta el momento (este último valor inicialmente es 0).
Luego, se recorre el arreglo que contiene los horarios de salida decremendo el contador de trabajadores dentro de la empresa, (simulando de ese modo el egreso de un trabajador). hasta encontrar un horario mayor al indexado en el arreglo de entradas, en cuyo caso se alternaba al arreglo a iterar.
Esta iteración .ªlternada"sobre los arrelgos se repite, hasta haberse indexado todas las posiciones del arreglo con los horarios de entrada, momento en el cual se devuelve el valor de la variable en la que se fueron almacenando los máximos.

A modo de una explicación más clara que nos acerque un poco más a la implementación, detallamos lo expuesto anteriormente a través del siguiente pseudocódigo:

**int maxCantProgJuntos((int,int) entradas[], (int,int) salidas[])**

[8]Cada trabajador es representado por un número natural

```
1  Complejidad: O(n)
2  var i,j,maxJuntos,juntosPorAhora : ℤ
3  var termine : bool
4  i ← j ← maxJuntos ← juntosPorAhora ← 0;                          // O(1)
5  termine ← false;                                                 // O(1)
6  while (!termine);                                                // O(n)
7  do
8  |  while (noLlegueAlFinal(i,n) && hora(entradas[i]) ≤
   |  hora(salidas[j]));                                           // O(1)
9  |  do
10 |  |  juntosPorAhora++ ;                                        // O(1)
11 |  |  i++ ;                                                     // O(1)
12 |  end
13 |  termine = i ≥ n ;                                           // O(1)
14 |  if (maxJuntos ≤ juntosPorAhora);                            // O(1)
15 |  then
16 |  |  maxJuntos ← juntosPorAhora;                               // O(1)
17 |  end
18 |  while (!termine && hora(salidas[j]) ≤ hora(entradas[i]));    // O(1)
19 |  do
20 |  |  juntosPorAhora−− ;                                        // O(1)
21 |  |  j++ ;                                                     // O(1)
22 |  end
23 end
24 return maxJuntos
```

### 3.3. Análisis de la complejidad del algoritmo

Para realizar el análisis de la complejidad del algoritmo, se decidió utilizar el modelo uniforme y no el logarítmico. Esto se debió a que en este caso, no resulta lógico evaluar la complejidad de acuerdo al costo de representar los valores de los parámetros de entrada. Más aún, por la forma y el contexto del problema y por el algoritmo implementado para la resolución, no sería correcto hacer un análisis logarítmico ya que en este caso las horas representables están acotadas (habíamos dicho que se encontraban entre 00:00:00 y 23:59:59) y la cantidad de trabajadores, aunque no está explícitamente acotada, se podría suponer como tal. De esta manera, resulta válido considerar que la complejidad espacial de cada elemento es unitaria, haciendo inadecuado analizar la complejidad del algoritmo con el modelo logarítmico.

Con el objetivo de realizar un mejor análisis de la complejidad del algoritmo propuesto, vamos a analizar el mismo remitiéndonos al pseudocódigo citado en la sección [3.2.1].

Observando de un modo más minucioso el pseudocódigo, se puede apreciar que la complejidad del algoritmo está dada por $n$, que hace referencia a la cantidad de trabajadores de la empresa. Si se observan las complejidades de cada operación, es facil advertir que todas son constantes, con excepción del flujo "while" más grande (aquél que tiene como condición: $(!termine)$).
Como se puede ver en la línea 5, el booleano *termine* es inicializado con el valor de verdad *false*. La otra variable que va a ser de interé para este análisis es $i$, la cuál se inicializa con el valor 0. Con estos valores, y sabiendo que el flujo while que se está analizando tiene como condición la negación de *termine*, se desprende que el algoritmo terminará sí y solo sí *termine* tome el valor de verdad *true*.
En la línea 13 del pseudocódigo es en el único lugar que se modifica el valor de verdad de *termine*. En esta línea se ve que:

$$termine \leftarrow i \geq n \tag{14}$$

Entonces, juntando esto con lo dicho anteriormente, se puede asegurar que el algoritmo va a finalizar una vez que $i \geq$ n. Veamos entonces cómo se comporta $i$ a lo largo del programa. La variable $i$ sólo es modificada en el primer while anidado, incrementándose de a uno por vez. Por lo tanto, una vez que se ejecute el contenido de este while anidado $n$ veces, el algoritmo finalizará.

Considerando la precondición que dice que todos los trabajadores ingresan a la empresa estrictamente antes de egresar, nos permite afirmar entonces que este primer while anidado va a ejecutarse siempre a lo sumo n veces[9]. Por lo tanto, el algoritmo tiene una complejidad de $O(n)$. Es más, se puede asegurar que el algoritmo nunca va a terminar antes de recorrer toda la lista de ingresos, por lo que el mejor caso es $\Omega(n)$, es decir, el algoritmo es $\Theta(n)$

A continuación se demuestra que la complejidad del algoritmo es lineal en el tamaño de la entrada.

Para empezar, consideremos $n =$ *cantidad de programadores de la empresa*. Se tiene entonces que:

- C(I) = $O(n)$
- E(I) = $O(n)$

En este caso, se puede ver claramente y sin necesidad de aplicar la definición de la relación entre la complejidad y el tamaño de la entrada, que:

$$C(I) = c' * E(I) \Leftrightarrow C(I) = O(E(I)) \tag{15}$$

### 3.4. Detalles de Implementación

Para compilar este programa, se debe referir a la carpeta que contiene al ejercicio nº 3 y ejecutar make en consola.

[9]Ver condición del primer while anidado

Luego de ejecutar make, se creara un archivo ejecutable, llamado main. Para ejecutarlo, se debe hacer una llamada desde la consola respetando el siguiente formato:
`main [nombreEntrada] [nombreSalida]`.

Los parámetros son opcionales, pero se puede elegir entre no pasar ningún parámetro, o pasar 2. Si no se pasan parámetros, el programa tomara los valores por defecto, esto quiere decir que el archivo de entrada será el que vienen incluído con la consigna del presente trabajo.

## 3.5. Debate

Se puede apreciar en los gráficos propuestos en la sección [4.3] que en general la implementación realizada se comporta como se esperaba que en teoría lo hiciera. Es decir, durante el desarrollo del informe sobre este ejercicio, entre otras cosas, se postuló la hipótesis de que la complejidad del algoritmo tenía un costo de $\Theta(n)$, donde $n$ es la cantidad de trabajadores de la empresa.

Se puede identificar en cada gráfico una marcada similitud entre los resultados arrojados en las distintas mediciones que se hicieron sobre la implementación y alguna recta (o función lineal), variando esta última su pendiente para cada caso. Esta pendiente podría compararse o considerarse como la constante que acompaña a la complejidad del algoritmo.

Se puede notar claramente que esta constante o pendiente, es mayor en aquellos gráficos en los cuales se evaluó el comportamiento del algoritmo en el peor caso que en los que se evaluó el mismo con entradas al azar.

En el segundo gráfico perteneciente al cuadro [10] podemos notar dos casos de outliers. Si observamos más detalladamente se aprecia que estos casos se dan con valores de $n$ grandes. Esto puede deberse a que, como el algoritmo tarda más en terminar[10], es más probable que haya alguna interrupción que irrumpa durante el procesamiento del mismo haciendo que la medición no sea precisa. Esto se debe a la forma en que se miden los ciclos de reloj. Lo que se hace es obtener los ciclos de reloj transcurridos hasta antes de hacer la llamada al algoritmo implementado, luego se toma la misma medición y, finalmente, se toma la diferencia entre ambas obteniendo como resultado la cantidad de ciclos insumidos por el algoritmo. Pero si durante el algoritmo, llega una interrupción y la atención de la rutina de la misma le insume al procesador varias instrucciones, esto puede llegar a provocar una medición de ciclos de reloj muy imprecisa.
Más aún, si tomamos en cuenta cómo un sistema operativo hace manejo de los procesos, puede haber llegado a ocurrir que durante la ejecución del algoritmo, este haya consumido todo su tiempo de ejecución asignado por el sistema, por lo que puede haber tenido que esperar una o incluso varias veces a que el sistema deje continuar con su ejecución.

## 3.6. Conclusiones

Luego de describir el funcionamiento del algoritmo, de realizar las pruebas y gráficos pertinentes y de analizarlos detalladamente, podemos realizar algunas conclusiones.

[10] recordar además que este gráfico corresponde al análisis en el peor caso

Podemos asegurar fehacientemente que la complejidad del algoritmo propuesto es lineal y que la misma es $\Theta(n)$ donde $n$ es la cantidad de trabajadores. Esto no sólo se desprende del análisis teórico realizado anteriormente, sino también de las sucesivas pruebas realizadas. Claramente se puede observar en todos los gráficos presentados cómo el comportamiento de la implementación se asemeja a una función lineal sobre los datos de entrada.

Finalmente, si hacemos una comparación entre los resultados obtenidos al aplicarse el algoritmo sobre datos de entrada azarosos y datos de entrada que se condicen con el peor caso, se puede observar que la constante que acompaña a la complejidad para el peor caso es considerablemente mayor a la constante que aparece en el caso azaroso. Como una conclusión poco menos significativa entonces, se puede decir que los casos en que los horarios de los trabajadores son todos disjuntos son los peores casos para este algoritmo.

# 4. Resultados y gráficos

## 4.1. Resultados (Ejercicio 1)

El programa *input_gen1* fue desarrollado para poder generar diversos casos de pruebas en los que $n$ adopte distintos valores[11]. A continuación se describen cuáles son esos casos y se explica brevemente que se esperaba observar en cada uno de ellos.

`a.- Casos con n entre 1.000.000 y 1000.000.000:`
Se pensó en este tipo de casos para poder evaluar el comportamiento del algoritmo frente a valores de $n$ muy grandes.

`b.- Casos con n entre 1 y 1.000.000:`
Se pensó en este tipo de casos para poder evaluar el comportamiento del algoritmo frente a valores de $n$ bastante acotados.

`c.- Casos con b multiplo de n , con n entre 1 y 1.000.000:`
Se pensó en este tipo de casos para poder observar la situación más favorable, en la cual el algortimo solo debe efectuar una operación que es la de ver si $b \ mod \ n \ = \ 0$ y en ese caso devolver 0.

`d.- Casos con n =` $2^k - 1$`, con k entre 2 y 30:`
Se pensó en este tipo de casos para poder observar el comportamiento del algoritmo en la situación más desfavorable. Esta es en la cual al realizar la división enterda del exponente de $b$ por 2 el resultado es siempre un número impar, por lo cual el algoritmo entra por en el último *else* que es el bloque de código con mayor número de operaciones.

`e.- Casos con n primo con n entre 1 y 1000.000.000:`
Se pensó en este tipo de casos para poder evaluar el comportamiento del algoritmo frente a valores de $n$ primos.

Mediante cada uno de estos casos se buscó estudiar el comportamiento del algoritmo y medir su complejidad real, valiéndose para ello del conteo de la cantidad de operaciones que realiza el algoritmo para resolver el problema. No obstante, previo a la experimentación, se formularon varias hipótesis en cuanto a qué era esperable observar en cada caso en particular:

1) En el caso `c`, dado que el algoritmo realiza 2 asignaciones y una única cuenta (la de calcular el resto de $b$ módulo $n$), resulta esperable que la complejidad real sea de valor constante 3.

2) Para los casos `a` y `b` sabemos que la complejidad teórica del algoritmo es $O(log_2(n))$. No obstante, como los *n's* del caso `a` son del orden de $10^9$, mientras que los *n's* del caso `b` son del orden de $10^6$ cabría esperar que la cantidad de operaciones para los *n's* del caso `a` sean mayores que los del caso `b`.

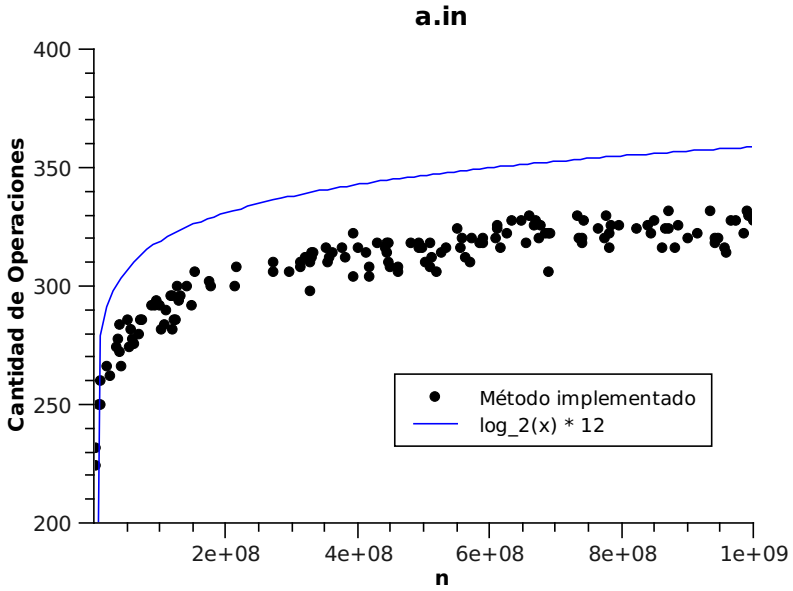
3) El caso de `e` es análogo al de `a`, ya que en ambos casos los *n's* son del orden de $10^9$, y la complejidad teórica del algoritmono varía. Sin embargo, se podría suponer a priori que por tratarse *n's* primos, la cantidad de operaciones del caso `e` sea ligeramente mayor a la cantidad de operaciones del caso `a`.

[11] Aclaración: En todos los casos b es un numero entre 0 y 1.000.000

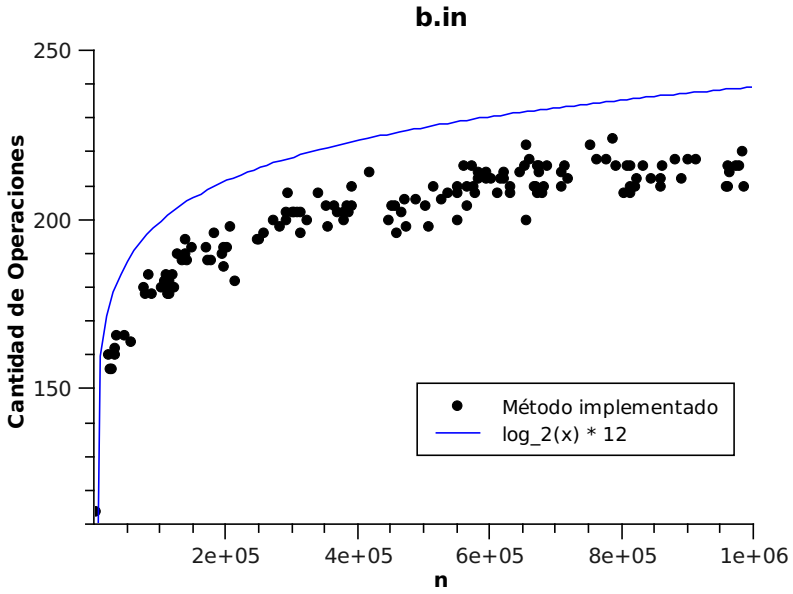
4) Por último, en el caso `d`, cada llamado recursivo del algoritmo siempre ingresa al blóque de código de los exponentes impares. Por ende, como realiza exactamente q operaciones $log_2$(n) veces es presumible que `d` sea el peor caso de este algoritmo.

Para la experimentación con el programa se generó un archivo con 150 pares de números $b,\ n \in \mathbb{N}$ para cada uno de los casos anteriores. Luego, se los procesó corriendo el programa *ejercicio_1* y cada archivo de entrada obtuvo su correspondiente archivo *"NombreDeArchivo.out"* y su correspondiente archivo *"NombreDeArchivo_grafico.out"* en el cual se registraron los valores de $n$ y la cantidad de operaciones que realizó el algoritmo en ese caso. Finalmente, haciendo uso de esos archivos se generaron, usando el programa de análisis gráfico *QtiPlot*, diversos gráficos de $n$ vs. *cantidad de operaciones* en los cuales se contrasta la curva de resultados con una función $f : \mathbb{N} \to \mathbb{R} / f(n) = c.log_2(n),\ c \in \mathbb{R}^+$ para poder estudiar si la complejidad real se ajusta a la complejidad teórica.

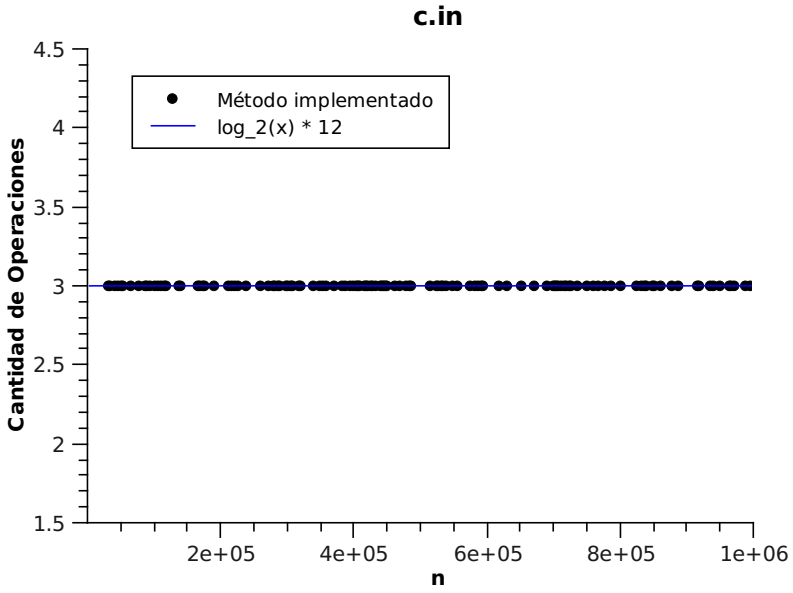
A continuación se presentan los gráficos realizados para cada caso:



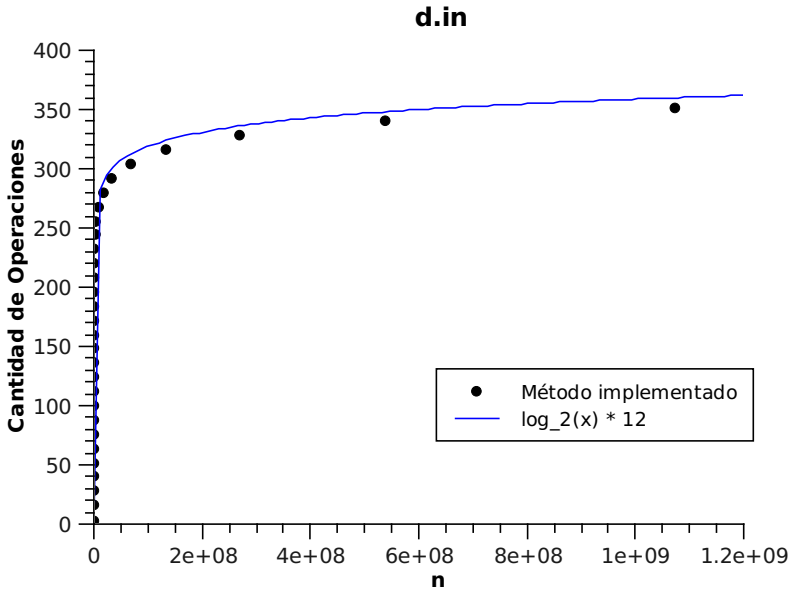

Cuadro 1: .



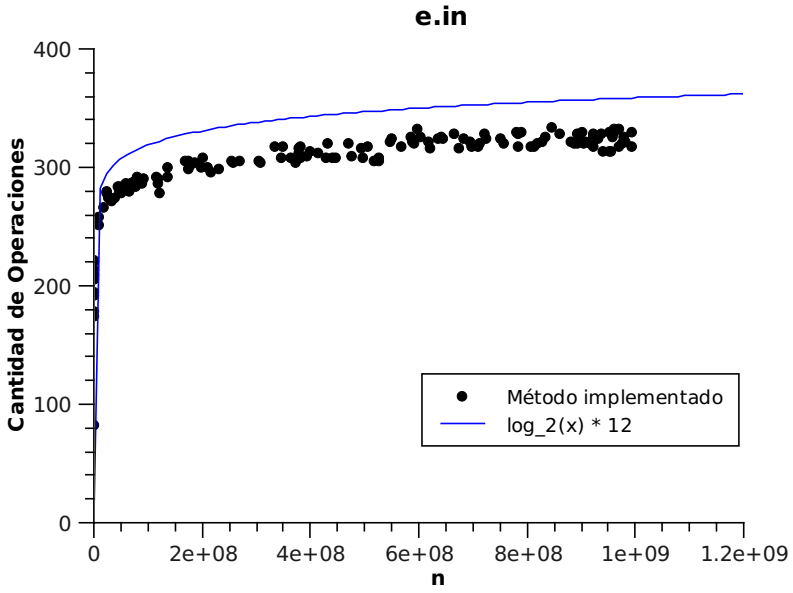

Cuadro 2: .




Cuadro 3: .

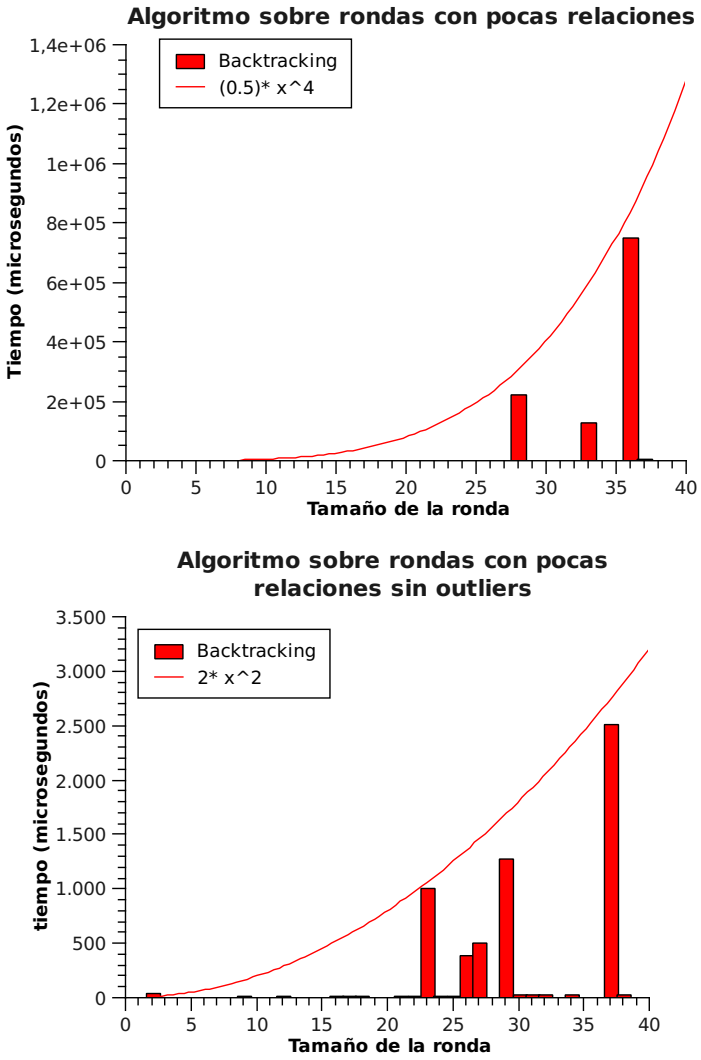



Cuadro 4: .

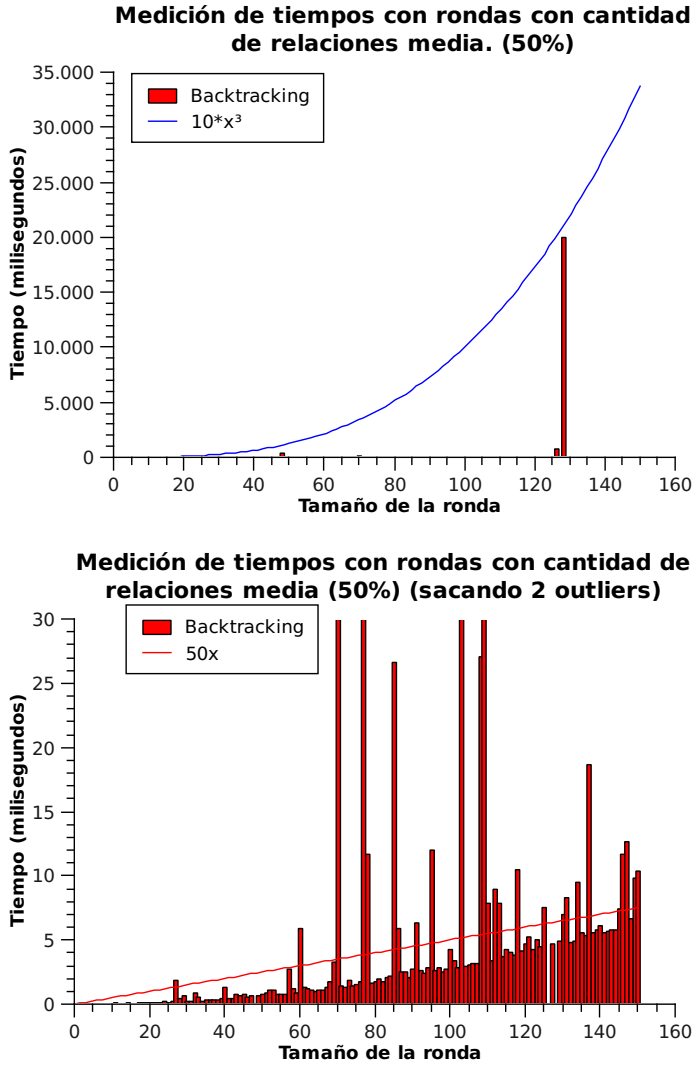



Cuadro 5: .

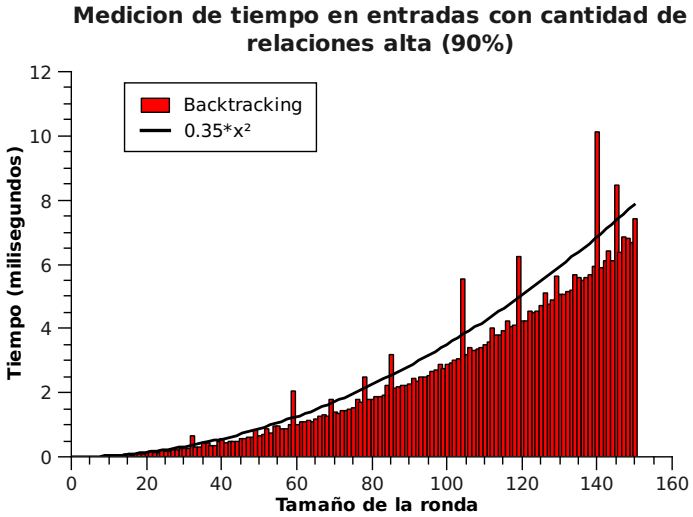
## 4.2. Resultados (Ejercicio 2)




Cuadro 6: Rondas con pocas relaciones: Estos graficos muestran la desigualdad entre tiempos de resolución para rondas es donde es complicado encontrar la solución. Se ve que quitando unos pocos outliers, los tiempos se encuentran en otros ordenes. Para ver este suceso, basta con obvservar las escalas de tiempo (en un gráfico se mueve entre 0 y 3500 microsegundos, y en la que posee outliers entre 0 y 1400000 microsegundos.) o también se puede observar las funciones que se utilizaron para dar cuenta de las dimensiones (en una comparada con $0{,}5 * x^4$ y en la otra con $2x^2$).




Cuadro 7: Rondas con una cantidad media de relaciones: En estos se puede observar como 2 outliers destruyen el caso promedio que se venia dando en el tiempo de resolución del algoritmo, pasando de graficos que se comparan con $10 * x^3$ contra otro que compara con $50 * x$, es decir, para rondas donde hay buenas probabilidades de que se pueda formar, el algoritmo parecería comportarse linealmente, pero claramente es una ilusión.



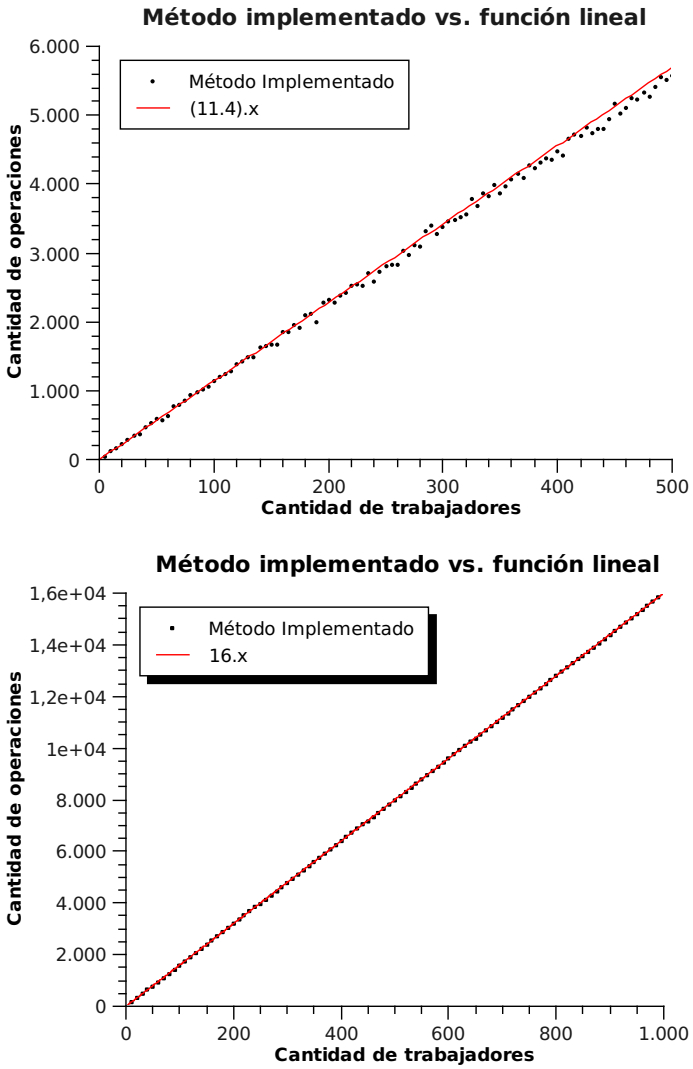

Cuadro 8: Rondas con una cantidad alta de relaciones: Cuando es muy poco probable que NO se pueda armar una ronda, el algoritmo y sus mejoras claramente funcionan bien (polimomial), como es el caso de este gráfico. Esto se debe a las propiedades del backtracking y a las mejoras implementadas.

### 4.3. Resultados (Ejercicio 3)

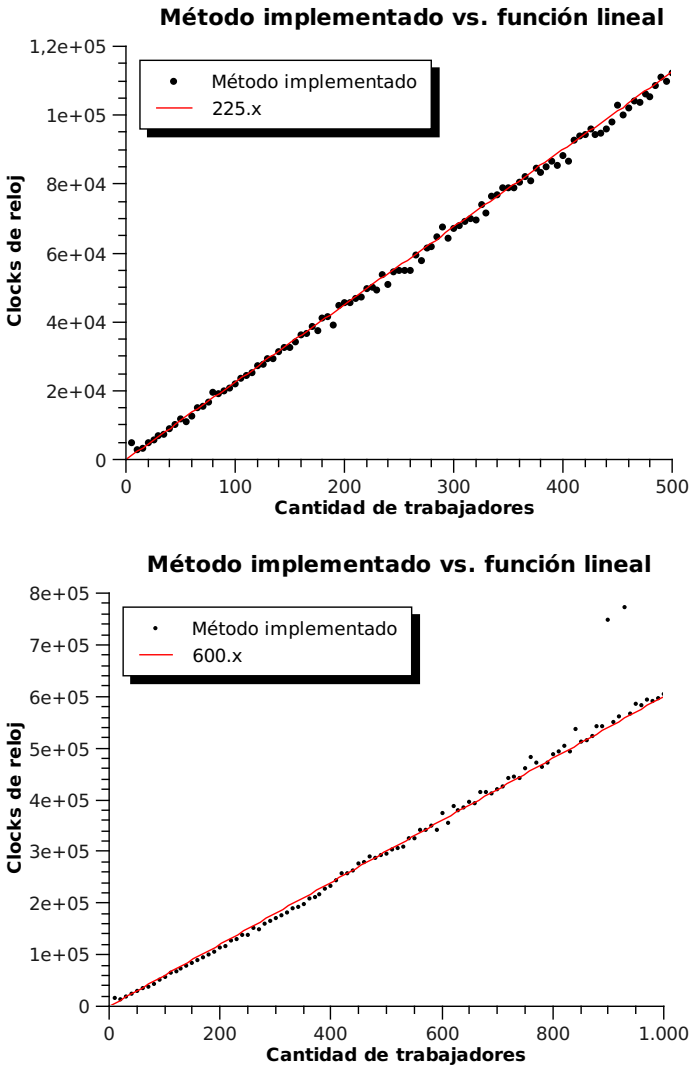
De la sección Explicación y de un análisis más minucioso del pseudocódigo, se desprende como algunas hipótesis que el peor caso se da cuando hay muchos trabajadores dispersos durante todo el día, es decir, si todos los horarios de salida y entrada entre dos trabajadores cualesquiera son disjuntos.

Para poder analizar el comportamiento del algoritmo implementado, se procedió a realizar un programa que genere distintas entradas, de acuerdo a distintos criterios que nos parecieron relevantes remarcar.
De esta manera, se utilizaron dos formas distintas de generar un archivo de entrada. Una de ellas se corresponde a una entrada completamente azarosa, a modo de analizar el comportamiento del algoritmo para una extensa cantidad de casos distintos. La segunda, trata de plasmar en sus casos la hipótesis de peor caso planteada en el párrafo anterior.

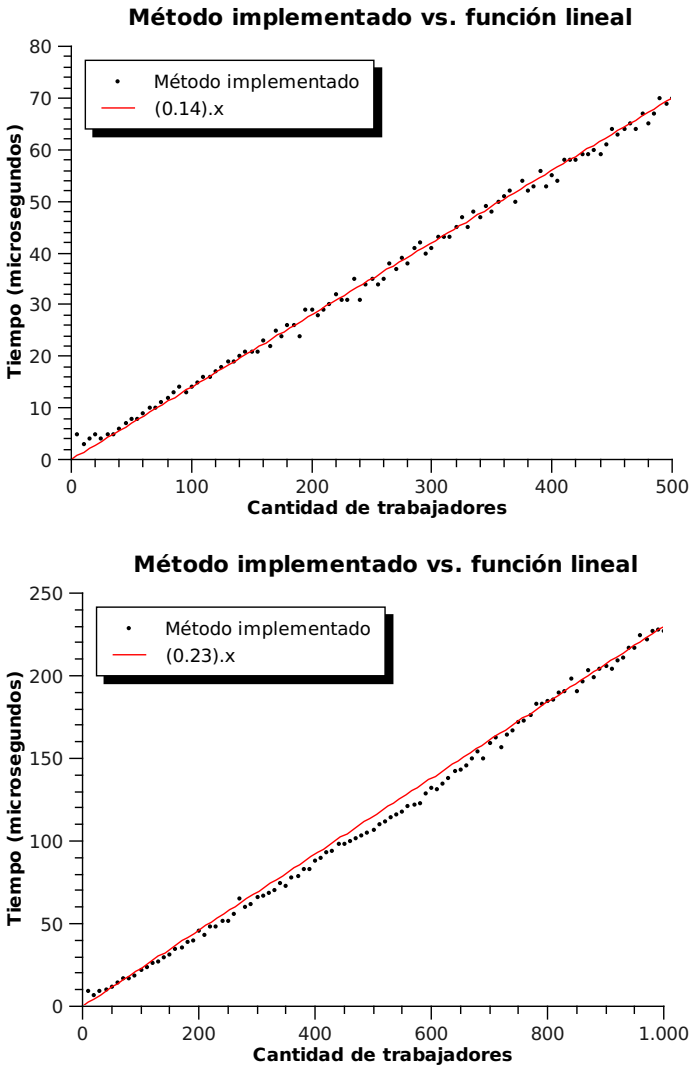
A continuación se adjuntan los gráficos provenientes de analizar los casos antes planteados. Están agrupados de a pares, a modo de mostrar más claramente la diferencia en la complejidad de cada caso.




Cuadro 9: Muestran el comportamiento de la cantidad de operaciones contra la cantidad de trabajadores. El primer gráfico proviene de una entrada hecha al azar. El segundo proviene de evaluar el comportamiento del algoritmo para el peor caso.




Cuadro 10: Muestran el comportamiento de la cantidad de ticks de reloj contra la cantidad de trabajadores. El primer gráfico proviene de una entrada hecha al azar. El segundo proviene de evaluar el comportamiento del algoritmo para el peor caso.




Cuadro 11: Muestran el comportamiento de la cantidad de microsegundos contra la cantidad de trabajadores. El primer gráfico proviene de una entrada hecha al azar. El segundo proviene de evaluar el comportamiento del algoritmo para el peor caso.